Human-Style Theorem Proving Using PVS*

Presented at TPHOLs '97, Murray Hill, NJ, August 19-22, 1997

Myla Archer and Constance Heitmeyer

Code 5546, Naval Research Laboratory, Washington, DC 20375
{archer,heitmeyer}@itd.nrl.navy.mil

Abstract. A major barrier to more common use of mechanical theorem provers in verifying software designs is the significant distance between proof styles natural to humans and proof styles supported by mechanical provers. To make mechanical provers useful to software designers with some mathematical sophistication but without expertise in mechanical provers, the distance between hand proofs and their mechanized versions must be reduced. To achieve this, we are developing a mechanical prover called TAME on top of PVS. TAME is designed to process proof steps that resemble in style and size the typical steps in hand proofs. TAME's support of more natural proof steps should not only facilitate mechanized checking of hand proofs, but in addition should provide assurance that theorems proved mechanically are true for the reasons expected and also provide a basis for conceptual level feedback when a mechanized proof fails. While infeasible for all applications, designing a prover that can process a set of high-level, natural proof steps for restricted domains should be achievable. In developing TAME, we have had moderate success in defining specialized proof strategies to validate hand proofs of properties of Lynch-Vaandrager timed automata. This paper reports on our successes, the services provided by PVS that support these successes, and some desired enhancements to PVS that would permit us to improve and extend TAME.

1 Introduction

Although the application of mechanical theorem provers to the verification of hardware designs has been somewhat successful, the use of such provers for verifying software is quite rare. A major barrier to more common use of mechanical theorem provers in both software and hardware verification, or verification of mathematical results in general, is the distance between the proof style natural to human beings and the proof style supported in various mechanical theorem provers.

Our goal is to make mechanical proof tools more useful to those who are not experts in one or more mechanical theorem provers. This group includes most mathematicians, algorithm designers, and industrial software and hardware developers. Our approach is to develop a tool on top of an existing theorem prover which reduces the distance between specifications and proofs natural to people and the specifications and proofs supported by existing theorem provers. In an ongoing case study, we have been developing the tool TAME [2, 1, 3] on top of the PVS environment [24, 20]. TAME can be used to specify and reason about Lynch-Vaandrager timed automata.

1.1 Some challenging questions

Reducing the distance between hand proofs and proofs generated mechanically is very difficult if not impossible to achieve in full generality, because different applications have their own specialized languages and conventions. This observation raises a number of questions:

* This work is funded by the Office of Naval Research. URLs for the authors are http://www.itd.nrl.navy.mil/ITD/5540/personnel/{archer,heitmeyer}.html

1. Can restricting the problem to specific application domains make the problem more manageable, and, if so, how large can the domain be?
2. Can a theorem prover specialized for an application domain be used directly by engineers?
3. Can a specialized theorem prover help with proof *search* as well as proof checking?
4. What is required of the underlying theorem prover to support natural proof steps tailored for a particular application domain?

We use our experience with TAME to address these questions.

1.2 Our approach

Our assumption in developing TAME is that the answer to the first question is positive. TAME, which supports the specification and mechanical verification of Lynch-Vaandrager timed automata, has been used to check a number of specifications and proofs developed by Lynch and her collaborators [10, 15, 14, 26]. The proofs checked with TAME apply human-style reasoning in the context of concepts specific to timed automata models. We believe that analogous combinations of human-style reasoning with model-specific concepts will make mechanization of verification natural for other mathematical models as well.

Our approach is not to build a new mechanical theorem prover from scratch but to build upon an existing prover that provides the basic features needed in a prover. Our goal is to support "human-style" reasoning in a particular mathematical model through an appropriate top layer to PVS. By human-style reasoning, we mean reasoning of the sort found in typical hand proofs. Such proofs usually include large proof steps, each of which corresponds to many small, detailed steps in a mechanized proof. Although mechanical provers can use powerful general tactics or strategies to take large steps, these steps rarely correspond to the steps that a human takes in reasoning. Our goal is to design the top layer so that large steps taken by the prover correspond closely to large steps taken by humans.

Supporting human-style proofs provides many benefits. First, a mechanical proof that corresponds closely to a hand proof provides documentation that allows a person who is expert in a given mathematical model, but not necessarily an expert in the use of a mechanical prover, to understand the proof and thus decide whether the property proved holds for the reasons expected. Second, the hand proof of a property can be used in a direct way to search for a mechanized proof and provides an opportunity for a person who is a domain expert but not an expert in the full mechanized proof system to do some proof checking. Finally, when a mechanized proof that is more natural fails, the prover can provide feedback at a high conceptual level explaining why the proof (or the associated specification) is in error.

Natural mechanized proof steps must not only be human-style; they must be *human-sized* as well. Humans can, of course, reason in tiny steps, but reasoning at too low a level can obscure the "big picture". For readability, human-sized steps are important. They are also important for the efficient creation of mechanized proofs.

Our current goal in TAME is to develop PVS strategies for proof steps that closely resemble the steps in hand proofs. Thus, we have concentrated so far on how to design the underlying theorem proving support for TAME, rather

than a high-level interface. In the process, we have identified a set of services that a programmable prover should provide to facilitate the mechanization of natural proof steps. In particular, we have found that higher-order features are critical for efficient implementation of human-style steps. PVS has some of these higher-order features. However, we have identified both additional higher-order features and other features currently lacking in PVS, that would increase the range of human-style steps that we could mechanize using PVS. Such features should prove useful not only to support TAME in PVS; they should be generally useful for supporting human-style proofs in any programmable prover.

1.3 Related efforts

Some approaches to mechanical verification are designed to be totally automatic. Such approaches include model checking [19, 12, 13] and the protocol analyzer described in [6]. Tools based on these approaches can indeed be useful, but requiring the assertion checking to be completely automatic limits their range of application.

Significant progress towards supporting human-style proofs in a mechanical prover has been made in the Mizar project [21]. As noted in [22], Mizar proofs tend to be very detailed, and unless care is taken in their construction, not easily read. The Mizar system has been used primarily to check proofs in pure mathematics; whether Mizar can be applied efficiently to verify software is not clear. In [9], Harrison shows how Mizar can be emulated in HOL. As a result, HOL-based proofs can use human-style reasoning that follows the Mizar formalities and required level of detail and thus can be easily understood. Using either Mizar or the HOL Mizar mode requires that one learn the details and conventions of a general theorem proving system before one uses the system in conjunction with a specialized mathematical model. This is the problem that TAME is intended to remedy.

Although different in their approaches from TAME, both the Mizar system [9] and Brackin's protocol analyzer [6] demonstrate how specialized theorem proving support can be built using a programmable theorem prover. The proof assistant for the Duration Calculus (DC) [25] also illustrates this approach. Proofs in the DC proof assistant are not human-style but use a Gentzen-style sequent proof system developed especially for the tool.

1.4 Preview of the paper

Section 2 provides technical background needed in the rest of the paper, while Section 3 describes the degree of success TAME has had in mimicking human-style reasoning. Section 4 illustrates the PVS features that have been useful in supporting human-style reasoning and discusses additional features desirable in the underlying prover. Section 5 discusses the features desirable in the underlying system to support the development of specialized theorem-proving, and Section 6 responds to the questions posed above. Finally, Section 7 discusses our plans for the further development of TAME.

2 Background

This section describes the mathematical model upon which TAME is based, and gives a brief introduction to both PVS and TAME.

2.1 The Lynch-Vaandrager Timed Automata Model

A Lynch-Vaandrager (LV) timed automaton is a very general automaton, i.e., a labeled transition system that incorporates the notions of current time and timed

transitions. In the model, a system is described as a set of timed automata, interacting by means of common actions. For verification purposes, these interacting automata can be composed into a single timed automaton. An automaton need not be finite-state: for example, the state can contain real-valued information, such as the current time, water level or steam rate in a boiler, velocity and acceleration of a train, and so on. This makes timed automata suitable for modeling not only computer systems but also real-world quantities, such as water levels and acceleration. LV timed automata can have nondeterministic transitions; this is particularly useful for describing how real-world quantities change as time passes.

The definition of timed automaton below, based on the definitions in [11, 10], was used in our case study involving a deterministic timed automaton [1].

A *timed automaton* A consists of five components:

- $states(A)$, a (finite or infinite) set of states.
- $start(A) \subseteq states(A)$, a nonempty (finite or infinite) set of start states.
- A mapping now from $states(A)$ to $R^{\geq 0}$, the non-negative real numbers.
- $acts(A)$, a set of actions (or events), which include special *time-passage* actions $\nu(\Delta t)$, where Δt is a positive real number, and *non-time-passage* actions, classified as *input* and *output* actions, which are *visible*, and *internal* actions;
- $steps(A) : states(A) \times acts(A) \rightarrow states(A)$, a partial function that defines the possible steps (i.e., transitions).

This definition describes a special case of Lynch-Vaandrager timed automata that requires the next-state relation, $steps(A)$, to be a function. By using the Hilbert choice operator ϵ, we are able to use essentially the same definition in the nondeterministic case as well [3]. A challenge is how to make reasoning about nondeterminism in TAME resemble human-style reasoning as closely as possible.

The properties of timed automata that one wants to prove fall into three classes: (1) state invariants, typically proved by induction; (2) simulation relations; and (3) ad hoc properties of certain execution sequences of a timed automaton. Proofs in both (1) and (2) have a standard structure with a base case involving start states and a case for each possible action. They are thus especially good targets for mechanization. Below, we define timed executions, reachability and invariants, and simulation relations.

Timed Executions. A *trajectory* is either a single state or a continuous series (i.e., an interval) of states connected by time passage events. A *timed execution fragment* is a finite or infinite alternating sequence $\alpha = w_0 \pi_1 w_1 \pi_2 w_2 \cdots$, where each w_j is a trajectory and each π_j is a non-time-passage action that "connects" the final state s of the preceding trajectory w_{j-1} with the initial state s' of the following trajectory w_j. A *timed execution* is a timed execution fragment in which the initial state of the first trajectory is a start state.

A timed execution is *admissible* if the total time-passage is infinity. The notion of admissible timed executions is important in expressing properties defined over time intervals (e.g., the gate at a railroad crossing is not down after a train has left unless a new train is about to arrive), rather than time points, and in defining simulation relations between timed automata.

Reachability and Invariants. A state of a timed automaton is *reachable* if it is the final state of the final trajectory in some finite timed execution of the automaton. An *invariant* of a timed automaton is any property true of all reachable states, or equivalently, any set of states containing the reachable states.

Simulation Relations. A *simulation relation* [18, 17, 16] relates the states of one timed automaton A to the states of another timed automaton B in such a way that the (visible) actions and their timings in admissible timed executions correspond. The existence of a simulation relation from A to B implies that each visible behavior (i.e., timed sequence of visible actions) of automaton A is a member of the set of visible behaviors of automaton B.

2.2 PVS

PVS (Prototype Verification System) [24] is a specification and verification environment developed by SRI. The system provides a specification language, a parser, a type checker, and an interactive proof checker. The PVS specification language is based on a richly typed higher-order logic. Proof steps in PVS are either *primitive* steps or *strategies* defined using primitive proof steps, applicative Lisp code, and other strategies. Strategies may be built-in or user-defined. Proof goals in PVS are represented as Gentzen-style *sequents*. To satisfy a proof goal, one must establish that the antecedent formulae imply one of the (zero or more) consequent formulae.

The primitive proof steps of PVS incorporate arithmetic and equality decision procedures, automatic rewriting, and BDD-based boolean simplification. Thus, PVS provides *both* a highly expressive specification language and automation of most low-level proof steps, in contrast to to other widely used proof systems, such as HOL [8] (which is lacking in decision procedures) and the Boyer-Moore theorem prover [5] (whose specification language is first-order). The programmability of PVS makes it a candidate to be the basis for specialized tools.

2.3 TAME

TAME (for Timed Automata Modeling Environment) is based upon a standard template specification for the timed automata described in Section 2.1, a set of standard theories, and a set of standard PVS strategies. The TAME template for specifying Lynch-Vaandrager timed automata provides a standard organization for an automaton definition. To define a timed automaton, the user supplies the following six components:

- declarations of the non-time actions,
- a type for the "basic state" (usually a record type) representing the state variables,
- any arbitrary state predicate that restricts the set of states (the default is **true**),
- the preconditions for all transitions,
- the effects of all transitions, and
- the set of start states.

In addition, the user may optionally supply

- declarations of important constants,
- an axiom listing any relations assumed among the constants, and
- any additional declarations or axioms desired.

To support mechanical reasoning about timed automata using proof steps that mimic human proof steps, TAME provides a set of standard strategies we have constructed using PVS. These strategies are based on a set of standard theories and certain template conventions. For example, the induction strategy, which is used to prove state invariants, is based on a standard automaton theory called **machine**. To reason about the arithmetic of time, we have developed a special theory called **time_thy** and an associated simplification strategy called **TIME_ETC_SIMP** for time values that can be either non-negative real values or ∞.

3 Successes with Human-Style Proving in TAME

We have successfully created PVS strategies for many of the human-style proof steps needed to verify properties of timed automata. We discovered these strategies by constructing PVS proofs that resembled the corresponding hand proofs as much as possible. In constructing the proofs, we used the standard PVS rules and strategies and then generalized the results. In several cases, additional PVS features would have allowed us to more closely follow human-style steps. Section 4 describes these additional features.

We have applied TAME to several problems: the Generalized Railroad Crossing (GRC) problem [10, 11], a timed version of Fischer's mutual exclusion algorithm [15], the Boiler Controller problem [14], and a Vehicle Control System example [26]. Most recently, we have used TAME to check some properties of a Group Communication Service [7].

As noted in Section 2.1, the interesting properties of timed automata fall into three classes. Each class has its own associated proof styles. State invariants are proved either by induction or directly using other state invariants. Proofs of simulation either have a case structure similar to induction proofs or are direct proofs combining other simulation results. Proofs of properties of timed executions are more ad hoc in their structure but employ certain specialized types of inference in their domain-relevant steps. We have had moderate success in developing human-style proof steps for properties in the first and third classes and have used these steps to obtain proofs. Our major strategy for state invariant proofs, discussed in more detail below, sets up induction proofs. Due to limitations in the PVS specification language, we have not yet developed an analogous strategy for simulation proofs. Currently, PVS does not allow us to define the simulation property at a useful level of abstraction. Future improvements to PVS are expected to remove this barrier.

Below, we provide more detail about our PVS strategies, the extent to which they support the translation of hand proofs into PVS proofs, and the benefits we have gained from their use.

3.1 Some PVS strategies that support human-style proof steps

Some existing strategies. Below, we describe some of the PVS strategies we have built into TAME. These support steps that are frequently found in hand proofs of properties of timed automata.

The induction strategy performs the (usually implicit) step in human reasoning about state invariants that converts induction over the number of transitions from a start state to a reachable state into a base case (for start states) and a case for each possible action that could lead to a transition. Besides making this conversion, the induction strategy completes the trivial proof branches and presents the user with the nontrivial base or action cases. The knowledge that the prestate and poststate are reachable is carried along in the action cases; this facilitates the application of previously proved state invariant lemmas in a proof. For each TAME application, the appropriate induction strategy must be compiled from the declaration of the non-time actions entered into the template specification.

The invariant-lemma strategy supports the application of state invariants to arbitrary states during a proof, the default state in an induction proof being the

prestate. When applied to a state whose reachability status has not been established, the reachability of that state is retained as a condition on the invariant for that state.

The precondition strategy simply invokes the specific precondition in an action case of an induction proof. (The full precondition may have other nontrivial components such as bounds on the time when the action can occur.)

The constant-facts strategy. This strategy introduces known facts about the constant "parameters" in a timed automaton description.

The strategy **TIME_ETC_SIMP** combines simple case-based reasoning and reasoning about time arithmetic, where time values can be nonnegative real numbers or ∞. The combination is particularly useful, since the definitions of the operations in time arithmetic are themselves case-based. This strategy is currently our best approximation to the human-style step "it is now obvious".

The strategy **USE_EPSILON** supports reasoning about nondeterministic components in the poststate. It introduces the constraints on a nondeterministic component on the main proof branch and forks a side branch for the existence proof entailed by the use of Hilbert's ϵ-axiom. It (inconveniently) requires as one of its arguments the domain type of the predicate to which ϵ is applied. Thus, there is room for improvement in this strategy. Nevertheless, we have found it very helpful in our applications involving nondeterminism (even though ϵ does not fit our needs for reasoning about nondeterminism precisely [3]).

The last-event and first-event strategies have proved helpful in ad hoc proofs about timed executions. The last-event strategy adduces, on the main proof branch, the last event before a point in the execution that possesses a certain property. The property, the point in the execution, and the name to be assigned to the last event must be supplied as arguments. A side proof branch is created in which one is obliged to prove the existence of some event that occurs prior to the given point in the execution and has the property. The first-event strategy is symmetrically analogous. The strategies are supported by the lemmas **last_event** and **first_event** shown in Figure 2 in Section 4.

The discretization strategy. The discretization strategy is also mainly useful in proofs about timed executions. It permits the leap (when justified) from reasoning about all states during an execution to reasoning about all states at the beginning of some trajectory of an execution. Again, an appropriate side proof branch is created, in which one must show that the property to be proved itself satisfies the property **trajectory_constant**: i.e., it is constant in any trajectory. Improvements and variations on this strategy are under consideration.

Some future strategies. Once those PVS enhancements described in Section 4 that are currently under way are complete, we plan to implement additional strategies. We describe a few examples of these strategies below.

The reachability strategy will determine and make known in the course of a TAME proof the fact that a particular state is reachable. This strategy will make the invariant-lemma strategy more powerful in the context of proofs concerning timed executions. While obvious to a human that any state in an admissible timed execution is reachable, some effort is required to introduce this fact into a mechanized proof, due partly to the variety of ways one can represent a state. The strategy will invoke several lemmas, apply the relevant one, and remove

Lemma 6.3. *In all reachable states of SystImpl, if Trains.r.status = I for any r, then Gate.status = down.*

Proof: Use induction. The interesting cases are *enterI* and *raise*. Fix r.

1. $enterI(r)$

   By the precondition, $s.Trains.r.status = P$.

   If $s.Gate.status \in \{up, going\text{–}up\}$, then Lemma 6.1 implies that $s.Trains.first(enterI(r)) > now + \gamma_{down}$, so $s.Trains.first(enterI(r)) > now$. But, the precondition for $enterI(r)$ is $s.Trains.first(enterI(r)) \leq now$. This means that it is impossible for this action to occur, a contradiction.

   If $s.Gate.status = going\text{–}down$, then Lemma 6.2 implies that $s.Trains.first(enterI(r)) > s.Gate.last(down)$. By Lemma B.1, $s.Gate.status = going\text{–}down$ implies $s.Gate.last(down) \geq now$. This implies that $s.Trains.first(enterI(r)) > now$, which again means that it is impossible for this action to occur.

   The only remaining case is $s.Gate.status = down$. This implies $s'.Gate.status = down$, which suffices.

2. *raise*

   We need to show that the gate doesn't get raised when a train is in I. So suppose that $s.Trains.r.status = I$. The precondition of *raise* states that $\exists r: s.CompImpl.r.sched\text{–}time \leq now + \gamma_{up} + \delta + \gamma_{down}$, which implies that, for all r, $s.CompImpl.r.sched\text{–}time > now$. But Parts 1 and 3 of Lemma 5.1 imply that in this case, $s.Trains.r.status = P$, a contradiction.

Fig. 1. A typical hand proof we have mechanized in TAME is the proof of the Safety Property from [11].

the irrelevant information. Its implementation will combine formula naming and recognition by content.

Naming strategies will keep the expanded versions of complex expressions—most notably, the representation of the poststate in an induction proof—out of sight, but retrieve, use, and hide their definitions when simplification strategies are applied to the sequent. The implementation of these strategies will combine formula naming and improved access to hidden formulae.

A skolemization-instantiation strategy will coordinate skolemization and instantiation of pairs of quantified formulae. This strategy can improve the induction strategy for state invariants that are quantified formulae. Its implementation will require the ability to probe for information about the number and type of the quantified variables in a formula. Its argument formulae will usually be identified by name.

The inductive-hypothesis strategy will retrieve the uninstantiated inductive hypothesis from among the hidden formulae, in induction proofs of quantified state invariants. It is needed in those rare cases where the default instantiation provided by the improved induction strategy is not the one desired. Its implementation will use formula naming and improved access to hidden formulae.

3.2 Translation support from our strategies

In the GRC, Fischer's Algorithm, the Boiler System, and the Vehicle Control Systems examples, we used TAME successfully to check both induction proofs and direct proofs of state invariants. We succeeded in mechanizing all of the state invariant proofs in these examples with the exception of a few induction proof branches involving complex arithmetic reasoning that could be checked by hand, but for which discovering the extra hints needed to supplement the decision procedures in PVS would be very time-consuming. In the majority of the induction proofs we mechanized, TAME's specialized strategies alone were enough to obtain the proofs. Figure 1 shows a typical hand proof from the GRC example. The steps in this hand proof are induction, appeal to invariant lemmas,

appeal to a precondition, and simple reasoning. Its mechanization uses only the induction strategy, the invariant-lemma strategy, the precondition strategy, and **TIME_ETC_SIMP**.

Other induction proofs sometimes required extra steps from the following four categories: (1) explicit substitutions to give the PVS decision procedures a boost, (2) manipulation to lift embedded quantified expressions to the top level, (3) expansion of a definition, and (4) application of lemmas about real arithmetic, sometimes accompanied by the use of the PVS CASE strategy to provide hints. They also occasionally involved what we consider a more legitimate application of the CASE strategy, namely, when reasoning by cases is a natural human proof step.

The degree of isomorphism between our TAME proofs and the hand proofs from which they were derived is very high for the GRC and Boiler System examples. For Fischer's Algorithm and the Vehicle Control Systems examples, the degree is less, but for different reasons. The hand proofs for Fischer's Algorithm used a different case breakdown than that supported by our induction strategy. In the Vehicle Control Systems example, the specification was in a form slightly different from the form TAME supports, and as a result the specification had to be translated into the required form. However, the TAME proofs did resemble the hand proofs in two important respects: the action cases considered significant by TAME and in the hand proofs were identical, and the facts required in the TAME proofs could be inferred from the hand proofs. Imposing some restrictions on the form of the human specification and proof would increase the degree of isomorphism.

3.3 Benefits of the TAME approach

We can construct induction proofs with TAME rather quickly. The initial capabilities of TAME were developed during our specification and verification of the GRC example. The next examples (with the exception of the Group Communication Service) took about two or three work weeks at most. Translating the specifications of each example into the TAME template required approximately two to three days. The proofs of individual invariants usually required approximately half an hour to half a day. Proofs of some of the more complicated invariants (whose hand proofs run from one to two and a half pages) often required two or three days to construct. The speed with which we can construct TAME proofs increases when we have a hand proof of similar structure as a guide. However, even in the absence of such a hand proof, or any hand proof at all, we have found TAME to be very helpful in proof exploration, because it simplifies the mechanization of many large steps natural in a hand proof.

In applying TAME to additional examples, we find that the previously developed strategies are highly reusable. This has contributed greatly to the speed with which we have been able to check new examples.

On occasion, a dead end was reached during an attempt to mechanize an induction proof using TAME. In the case of the original Boiler Controller specification, these dead ends revealed some errors in the specification and two of the proofs. Due to the form of the proofs, the contents of the sequents at these dead ends were easily traced to the specification and the point reached in the reasoning. Thus, the type of feedback we were able to provide in the Boiler Controller example was very specific in pinpointing both typographical and rea-

soning errors. In the case of the Group Communication Service, we are using TAME for proof exploration as well as proof checking. Thus, we cannot always anticipate the line of reasoning that will succeed nor the meaning of dead ends in a proof. In some cases, a dead end suggested the reformulation of complex invariants. In others, dead ends have uncovered additional invariants needed in the full correctness proof.

4 Desirable Capabilities in the Underlying Prover

In developing PVS strategies for human-style proofs, we have found certain PVS features to be particularly helpful and have discovered other features, currently missing in PVS, that would be helpful if provided. The features that we have identified should be useful for supporting human-style proof steps in other programmable theorem provers—not just PVS.

Useful higher order features of PVS. To define several of the generic steps, we found various higher-order features of PVS useful. The most useful higher-order feature is the ability to quantify over predicates in definitions and lemmas. This feature permits us (see Figure 2) to state the induction principle **machine_induct**, the existence lemmas **last_event** and **first_event**, and the definitions of the function **discretize** and related concepts involved in the discretization strategy. This feature would also permit support for reasoning that uses such higher-order concepts from real analysis as "convex function", which

Example 1: The theorem **machine_induct** and supporting definitions.

```
base(Inv) : bool = (FORALL s: start(s) => Inv(s));
inductstep(Inv) : bool = (FORALL s, a: reachable(s) & Inv(s) & enabled(a,s) => Inv(trans(a,s)));
inductthm(Inv): bool = base(Inv) & inductstep(Inv) => (FORALL s : reachable(s) => Inv(s));
machine_induct: THEOREM (FORALL Inv: inductthm(Inv));
```

Example 2: The lemmas **last_event** and **first_event** with supporting definitions.

```
state_event_prop: TYPE = [atexecs,states,posnat -> bool];
Q: state_event_prop;
last_event: LEMMA (FORALL (alpha:atexecs, s:states, P:state_event_prop):
  (LET Q = (LAMBDA(alpha:atexecs, s:states, n:pos_nat): (precedes_state(alpha)(n,s) & P(alpha,s,n))) IN
   (FORALL (n:posnat): (Q(alpha,s,n) => (EXISTS (m: posnat): m >= n & Q(alpha,s,m)
                                        & (FORALL (k: posnat): k >= m & Q(alpha,s,k) => k = m))))));
first_event: LEMMA (FORALL (alpha:atexecs, s:states, P:state_event_prop):
  (LET Q = (LAMBDA(alpha:atexecs, s:states, n:pos_nat): (precedes_event(alpha)(s,n) & P(alpha,s,n))) IN
   (FORALL (n:posnat): (Q(alpha,s,n) => (EXISTS (m: posnat): m <= n & Q(alpha,s,m)
                                        & (FORALL (k: posnat): k <= m & Q(alpha,s,k) => k = m))))));
```

Example 3: The theorem **discrete_equiv** and some supporting definitions.

```
state_pred: TYPE = [atexecs -> [states -> bool]];
index_pred: TYPE = [atexecs -> [nat -> bool]];
discretize (T: state_pred): index_pred =
  LAMBDA (alpha:atexecs): LAMBDA (n:nat): T(alpha)(fstate(w(alpha)(n)));
trajectory_constant(SP:state_pred): bool = FORALL (alpha:atexecs, s:states):
  (in_atexec(alpha)(s) => (SP(alpha)(s) = SP(alpha)(fstate(w(alpha)(traj_index(alpha)(s))))));
discrete_equiv_pred(T:state_pred):bool = (trajectory_constant(T) =>
  (FORALL (alpha:atexecs, s:states):
     (in_atexec(alpha)(s) => (T(alpha)(s) = (discretize(T))(alpha)(traj_index(alpha)(s))))));
discrete_equiv: THEOREM FORALL (T:state_pred): discrete_equiv_pred(T);
```

Fig. 2. Some higher-order definitions useful in supporting human-style proof steps.

turned up in the Boiler Controller example. Whether adding such support to TAME will prove worthwhile is at this point unknown.

The ability to define the state of an automaton as a record, some of whose components are functions, supports our strategies in a more indirect way, by simplifying the template conventions followed in specifying a particular automaton and relied upon by our strategies.

Other useful PVS features. Another feature of PVS useful for supporting human-style proof steps is the presence of built-in decision procedures. These decision procedures provide much of the support needed (for example, in our strategy **TIME_ETC_SIMP**) for taking the analogous mechanical steps in the many cases where a hand proof contains steps such as "it is obvious", "this is a contradiction", etc. The existing decision procedures handle propositional logic, equational reasoning, automatic rewriting, and linear arithmetic and can be invoked concurrently using built-in PVS strategies.

Desirable higher-order features lacking in PVS. PVS lacks some useful higher-order features. One such feature is parametric polymorphism, which permits the types of the parameters in definitions or lemmas to be type variables or to include type variables in their representation. Another higher-order capability missing in PVS is one that would permit us to express the definition of a simulation relation between automata. This capability is discussed in more detail below.

Parametric polymorphism requires on-the-fly type inference, but can simplify both specification and proof. Currently, the following cannot be done in PVS without explicit type information from the user:

- definition of a predicate stating that the value of a state component is unchanged by time passage events;
- application of generic lemmas on parameterized types, such as queues and ordered lists;
- definition of generic lemmas that support strategies for converting $\forall$ expressions to $\neg\exists\neg$ expressions and, similarly, $\exists$ to $\neg\forall\neg$; and
- invocation of the Hilbert ϵ-axiom on a predicate.

Given parametric polymorphism, we could eliminate many instances where application-specific detail is required, and extend the set of generic human-style proof steps supported by TAME. PVS does support generic definitions, axioms, and lemmas by way of parameterized *theories*. However, using these in a specification currently requires either providing explicit type information in places or explicitly importing all relevant theory instantiations. In either case, explicit type information is needed when the axioms or lemmas are invoked in the course of a proof. The need for such type information is a barrier to creating generic strategies that are simple to apply. For example, our strategy **USE_EPSILON** (see Section 3.1) currently needs as an argument not only the predicate to which the ϵ-axiom is being applied, but the domain type of that predicate.

When we try to define a simulation mapping generically in PVS (see Figure 3), we encounter dead ends. For example, to refer to an automaton by name in a definition, we need an automaton type. The obvious representation of an element of this type is a record whose components are the states, actions, start state predicate, transition function, and so on. However, states and actions are most naturally represented as types, and record components (unlike parameters

Let A and B be timed automata, and I_A and I_B be, respectively, state invariants of A and B. Let f be a binary relation between $states(A)$ and $states(B)$. Then f is a *simulation mapping* from A to B if it satisfies the following three conditions:

1. If $u \in f[s]$ then $now(u) = now(s)$.
2. If $s \in start(A)$ then $f[s] \cap start(B) \neq \emptyset$.
3. If $s \xrightarrow{\pi}_A s'$ is a step of A, $s, s' \in I_A$, and $u \in f[s] \cap I_B$, then there exists $u' \in f[s']$ such that there is a timed execution fragment from u to u' having the same timed visible actions as the step π.

Fig. 3. Definition of simulation between automata. The notation $f[s]$ stands for $\{u : (s, u) \in f\}$.

of theories) cannot have type "type" in PVS. All the alternative, less natural solutions we have considered also seem to require some feature currently missing in PVS.[1]

Other desirable features lacking in PVS. Several PVS features are being added. These will permit us to implement several human-style strategies for which we have designs. These features include the ability to name formulae and subformulae, to retrieve a hidden formula by name, and to identify formulae by content. They will permit the writing of strategies that can use specific formulae in a sequent without referring to their exact "address". The usefulness of such a feature in HOL has been noted by others [4]. Section 3.1 outlined some of our intended applications of these features.

We have also identified other features, some available in other mechanical provers, that would be extremely useful in supporting human-style reasoning steps. Examples include the ability to skolemize or instantiate embedded quantifiers, the expansion of the scope of the existing decision procedures to handle some obvious facts about nonlinear real arithmetic, better automated support for reasoning about expressions involving the constructors or destructors of an abstract data type, and the ability to do resolution-style reasoning with respect to all or part of a sequent. Below, we describe how these features would prove useful.

Embedded quantifiers. We have encountered many cases where a human reasoning step is sidetracked in a PVS proof because embedded quantifiers cannot be "reached" for skolemization or instantiation. Figure 4 illustrates two such cases. The first case in Figure 4 is a generic illustration of a situation in which instantiation of the quantified variables *in situ* would lead to a proof without splitting the current goal. The second case is an actual invariant lemma from the proof of the implementation of Fischer's Algorithm from [15], in particular, the Strong Mutual Exclusion invariant that guarantees that no two processes can simultaneously be in the critical section. In this case, the skolem constant of the embedded "FORALL" in the inductive conclusion must be used as the instantiation of the corresponding "FORALL" in the inductive hypothesis in nearly every proof branch. The structure of the mechanical proof is obscured by the need to extract the embedded quantified subexpressions. For both examples in Figure 4, the embedded quantifier problem could be dealt with by recasting the formulae as formulae quantified at the top level. For complex state invariants, however,

[1] In a future version of PVS, SRI plans to support both theory parameters for theories and theory interpretations [23]. For our purposes, either may be sufficient.

Example 1:

```
P(a)
(EXISTS (x): P(x)) => (FORALL (y): Q(y))
----------------------------------------------------------
Q(b)
```

Example 2:

```
Inv_5_4(s: states): bool = (FORALL (i: Index):
    (pc(i,s) = leave_trying OR pc(i,s) = critical OR pc(i,s) = reset) =>
       (user?(x(s)) & (name(x(s)) = i)
    & (FORALL (j: Index): ((NOT (j = i)) =>
           (NOT (pc(j,s) = set OR pc(j,s) = leave_trying OR pc(j,s) = critical OR pc(j,s) = reset))))));

lemma_5_4: LEMMA ( FORALL (s: states): reachable(s) => Inv_5_4(s) );
```

Fig. 4. A sequent and invariant with embedded quantifiers. The invariant *Inv_5_4* of *Lemma_5_4* embodies the Strong Mutual Exclusion property that implies that processors with different indices cannot simultaneously have their program counters in their critical region.

this cannot always be done; see, e.g., [7]. Even when recasting is possible, the resulting formulae are often less natural, and this violates the philosophy behind TAME.

Extended arithmetic decision procedures. In applying TAME to hybrid automata, we have encountered reasoning about nonlinear real arithmetic where PVS requires an interactive boost in the form of appropriate application of several lemmas about real arithmetic. Figure 5 shows the lemmas we have needed in practice. This list suggests useful extensions to the decision procedures.

Abstract data type reasoning. When an abstract data type is involved in a proof, many relevant inferences are obvious to a human but not to the prover. For example, in reasoning about the abstract data type **time** (see Figure 6), a human seeing $A = dur(B)$ immediately interprets this equality as equivalent to $fintime(A) = B$, but the prover requires detailed human guidance to make the same inference. Of course, if a data type constructor requires more than one argument, information about all the destructor-values are needed in drawing a conclusion about the constructor-value. A resolution-style step could be useful in handling this.

```
real_thy: THEORY
  BEGIN
    nonnegreal:TYPE = {r:real | 0 <= r};
    sq(x:real):real = x*x;
  % posreal_mult_closed: LEMMA (FORALL (x,y:real): (x > 0 & y > 0) => x*y > 0);
    nonnegreal_mult_closed: LEMMA (FORALL (x,y:real): (x >= 0 & y >= 0) => x*y >= 0);
    greater_eq_nonnegmult_closed: LEMMA (FORALL (x,y,z:real): (x >= 0 & y >= z) => x*y >= x*z);
    square_nonneg: LEMMA (FORALL (x:real): (sq(x) >= 0));
    nonpos_neg_quotient:LEMMA (FORALL (x:real,y:real): (x <= 0 & y < 0) => x/y >= 0);
    nonneg_pos_quotient:LEMMA (FORALL (x:real,y:real): (x >= 0 & y > 0) => x/y >= 0);
  END real_thy
```

Fig. 5. Some lemmas about real arithmetic

```
time: DATATYPE
 BEGIN
  fintime(dur:{r:real|r>=0}): fintime?
  infinity: inftime?
 END time
```

Fig. 6. The type **time** is a simple example of an abstract data type in PVS. It is the union type of nonnegative real numbers and ∞. The destructor *dur* extracts the argument of the constructor *fintime*. *fintime?* and *inftime?* are recognizer predicates.

Resolution-style steps. Another application we have encountered for a resolution-style step is in automatically proving that the value of a predicate is constant over a trajectory from its propositional structure and the properties of its atomic parts. This would be helpful in automating the proof obligation that would accompany the use of our discretization strategy.

5 Desirable Features in the Prover Environment

In developing specialized strategies in a programmable theorem prover, additional support beyond the services provided by the prover itself is highly desirable. Since strategy development involves much interactive experimentation with proofs, features that would allow the developer to experiment more efficiently are especially important. These features include efficient ways to save and test alternate proofs, efficient ways to continue incomplete branches of partial proofs without executing the remaining branches, and the ability to obtain timing information to help locate inefficient steps in a strategy under development (e.g., why does carrying along the reachability of the poststate in our induction strategy double its execution time?). The features mentioned are among the planned PVS enhancements.

6 Lessons learned

Based on our experience with TAME, we have found at least partial answers to the four questions listed in Section 1.1. Section 4 addresses the fourth question. Below, we address the first three.

Question 1. Restricting the domain does make a significant difference in the extent to which human-style theorem proving can be supported. In a restricted domain in which a limited repertoire of proof steps is sufficient for most human proofs, we have been able to provide substantial support for human-style mechanical proofs. Many of the human-style steps we support, such as the induction strategy, the precondition strategy, and the invariant-lemma strategy, are highly specific to the mathematical model supported by TAME, and are effective only when applied to specifications of a highly restricted form. The induction strategy is quite complex and very finely tuned for its purpose. Thus, we do not expect our techniques to be useful in supporting human-style proofs across the board. With respect to how large the restricted domain can be, we note that TAME's domain is significantly larger than that of other proof tools aimed at verifying automata—particularly model checkers, which require finite-state automata.

Question 2. So far, TAME has not been used by any engineers, so we still have no answer to this question. However, we have demonstrated that it is possible for a theorem proving expert to provide useful feedback to practitioners.

Question 3. Tackling the Group Communication Service problem has provided us with some experience in using TAME to search interactively for a proof. We have succeeded in constructing several induction proofs without the aid of a hand proof. Using TAME to keep track of the details and to identify the significant cases has helped in focusing the interactive contribution required in a search on determining which previously proved state invariants to apply. Making the current proof goals readable would be a significant help during proof search. In this, the naming strategies planned by SRI can play an important role.

7 Future Directions

The TAME proofs that we have produced have a structure which reflects the human-style proof steps that comprise them. However, what these human-style steps are is currently discernible only to a TAME expert. We plan to make TAME proofs more readable by non-experts in two ways, both of which require the enhancements to PVS in progress. First, we plan to name the TAME proof steps uniformly. Since some strategies, such as the induction strategy, are compiled for each particular automaton, doing this may require the ability to set and access a global variable through PVS whose value is the automaton name. Second, we plan to print, in the form of comments, the information introduced in certain proof steps, such as application of an invariant lemma or invocation of the precondition. To do this, we need the ability to retrieve the content of a named formula in the sequent.

Currently, we must compile the automaton-specific strategies by hand from the user's description of the nontime actions and a user-defined set of abbreviations. We plan to automate this procedure, either internally or externally to PVS.

The PVS enhancements we have discussed were inspired by our experience in developing TAME. We hope to use our continuing experience with the design of strategies for human-style reasoning and the development of specialized tools based upon PVS to motivate the further development of PVS, and perhaps other theorem proving systems, in what we consider to be an important direction.

Acknowledgments

We wish to thank N. Lynch, G. Leeb, V. Luchangco, H. B. Weinberg, A. Fekete, A. Shvartsman, and R. Khazan for providing us with challenging examples for testing TAME. We also thank R. Jeffords and S. Garland for helpful discussions.

References

1. M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proc. 1996 IEEE Real-Time Technology and Applications Symp. (RTAS'96)*. IEEE Computer Society Press, 1996.
2. M. Archer and C. Heitmeyer. TAME: A specialized specification and verification system for timed automata. In *Work-In-Progress Proc. 1996 IEEE Real-Time Systems Symp. (RTSS'96)*, pages 3–6, 1996.
3. M. Archer and C. Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. In *Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *Lect. Notes in Comp. Sci.*, pages 171–185. Springer-Verlag, 1997.
4. P. Black and P. Windley. Automatically synthesized term denotation predicates: A proof aid. In *Higher Order Logic Theorem Proving and Its Applications (HOL'95)*, volume 971 of *Lect. Notes in Comp. Sci.*, pages 46–57. Springer-Verlag, 1995.

5. R. Boyer and J. Moore. *A Computational Logic*. Academic Press, 1979.
6. S. Brackin. Deciding cryptographic protocol adequacy with HOL. In *Higher Order Logic Theorem Proving and Its Applications (HOL'95)*, volume 971 of *Lecture Notes in Computer Science*, pages 90–105. Springer-Verlag, 1995.
7. A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. In *Proc. Sixteenth Ann. ACM Symp. on Principles of Distributed Computing (PODC'97)*, pages 53–62, Santa Barbara, CA, August 1997.
8. M. J. C. Gordon and T. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
9. J. Harrison. A Mizar mode for HOL. In *Proc. 9th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lect. Notes in Comp. Sci.*, pages 203–220. Springer-Verlag, 1996.
10. C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A case study in formal verification of real-time systems. In *Proc., Real-Time Systems Symp.*, San Juan, Puerto Rico, Dec. 1994.
11. C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A case study in formal verification of real-time systems. Technical Report MIT/LCS/TM-51, Lab. for Comp. Sci., MIT, Cambridge, MA, 1994. Also Technical Report 7619, NRL, Wash., DC 1994.
12. T. Henzinger and P. Ho. Hytech: The Cornell Hybrid Technology Tool. Technical report, Cornell University, 1995.
13. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: the Automata-Theoretic Approach*. Princeton University Press, 1994.
14. G. Leeb and N. Lynch. Proving safety properties of the Steam Boiler Controller: Formal methods for industrial applications: A case study. In J.-R. Abrial, et al., eds., *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, vol. 1165 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1996.
15. V. Luchangco. Using simulation techniques to prove timing properties. Master's thesis, Massachusetts Institute of Technology, June 1995.
16. N. Lynch. Simulation techniques for proving properties of real-time systems. In *REX Workshop '93*, volume 803 of *Lecture Notes in Computer Science*, pages 375–424, Mook, the Netherlands, 1994. Springer-Verlag.
17. N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. To appear in *Information and Computation*.
18. N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. In *Proc. of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, pages 397–446. Springer-Verlag, 1991.
19. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
20. S. Owre, N. Shankar, and J. Rushby. User guide for the PVS specification and verification system (Draft). Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 1993.
21. P. Rudnicki. An overview of the MIZAR project. In *Proc. 1992 Workshop on Types and Proofs for Programs*, pages 311–332, June 1992. Also available through anonymous ftp as pub/cs-reports/Bastad92/proc.ps.Z on ftp.cs.chalmers.se.
22. P. Rudnicki and A. Trybulec. A note on "How to Write a Proof". In *Proc. 1992 Workshop on Types and Proofs for Programs*, June 1996. Available through P. Rudnicki's web page at http://www.cs.ualberta.ca/~piotr/Mizar/.
23. J. Rushby. Private communication. NRL, Jan. 1997.
24. N. Shankar, S. Owre, and J. Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 1993.
25. J. Skakkebaek and N. Shankar. Towards a duration calculus proof assistant in PVS. In *Third Intern. School and Symp. on Formal Techniques in Real Time and Fault Tolerant Systems, Lect. Notes in Comp. Sci. 863*. Springer-Verlag, 1994.
26. H. B. Weinberg. Correctness of vehicle control systems: A case study. Master's thesis, Massachusetts Institute of Technology, February 1996.

This article was processed using the LaTeX macro package with LLNCS style